Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 22 pages

Operational Subsumption, an Ideal Model of Subtyping

Laurent Dami [1]

Centre Universitaire d'Informatique
Université de Genève
Genève, Switzerland

Abstract

In a previous paper we have defined a semantic preorder called *operational subsumption*, which compares terms according to their error generation behaviour. Here we apply this abstract framework to a concrete language, namely the Abadi-Cardelli object calculus. Unlike most semantic studies of objects, which deal with typed equalities and therefore require explicitly typed languages, we start here from a untyped world. Type inference is introduced in a second step, together with an ideal model of types and subtyping. We show how this approach flexibly accommodates for several variants, and finally propose a novel semantic interpretation of structural subtyping as embedding-projection pairs.

1 Introduction

In a previous paper [10] we have defined a semantic preorder called *operational subsumption*, which compares terms according to their error generation behaviour. Together with the technical device of *labeled reductions*, used as a syntactic characterization of finite approximations, this semantics was shown to adequately interpret recursive types and subtyping. In this paper we apply this approach to **FOb**, the lambda-calculus of objects of Abadi and Cardelli [2]. Because we work with a concrete language instead of an abstract framework, several steps can be simplified, so the resulting semantic structure is intuitively quite obvious. Moreover, a "context lemma" in the untyped language gives us a simple induction principle for proving many properties of types. The goal is to show that the "Coverage of Operational Semantics" [21] can be widened to also deal with subtyping systems. More concretely, we

[1] work partially supported by Swiss SPP grant 5003-045332 and FNRS grant 2000-047181.96

show several directions where this approach can simplify or deepen previous results.

First, we give an interpretation of second-order bounded quantification for universal and existential types. This extends previous work on ideal models [18] with subtyping. Furthermore it answers to Abadi et al [3], who wondered whether their approach would apply to a suitable notion of operational ideals: this is exactly what is done here.

Then we have a direct way of interpreting typed equivalences of object-calculi (equivalences which depend on the type context in which objects are considered). Gordon and Rees [11] used the coinduction principle of Howe [14] to interpret these equivalences; however this required a heavy apparatus which switches between typed and untyped worlds: in addition to the reduction relation they had to define a labeled transition system and a "compatible refinement" relation. By contrast our interpretation is based on the untyped reduction relation. Moreover we can validate second-order typed equivalences, which remained an open issue in [11].

Finally we give a semantic interpretation of structural subtyping, which is useful for solving the problem known as "polymorphic object update". Bruce and Longo [7] have demonstrated that in usual interpretations of subtypes as subsets the polymorphic type $\forall X \leq T.X \to X$ can only contain the identity function, which makes it impossible to type some elementary updating operations on objects. The problem is developed in more detail in Chapter 16 of [2]. Some authors [13,20] have proposed to solve the problem by restricting the subtype relation in various ways so as to ensure that the subtypes have the same structure as the supertype. Here we show that usual subtyping and structural subtyping are two distinct notions semantically. The former corresponds to a subset relation, while the second corresponds to embedding-projection pairs in the subsumption ordering. Both subtyping notions can cohabit and could be included in the type syntax if so desired.

2 The untyped object calculus

The syntax shown in Figure 1 is built from the set ω of natural numbers, from a countable set $\mathcal{N}$ of *names* (for object fields) and a set $\mathcal{X}$ of *variables*. ε is a constant for errors. The main difference from [2] is that terms are *labeled*, i.e. decorated at each subterm with a natural number or ∞. Here the purpose of labels is merely to introduce a notion of finite projection for interpreting recursive types. In [10] we also used labels for an abstract definition of erroneous terms; this is not needed here, since we work in a concrete calculus for which the erroneous terms are just those which reduce to the error constant. Intuitively, each label acts as a counter limiting the number of interaction steps between the corresponding subterm and its context. When a label reaches 0, it becomes a divergent term, with which no interaction is possible. The infinite label ∞ imposes no limit, so for better readability it will usually be

$$
\begin{array}{lrcl}
\text{(indexes)} & i,j,k \in \omega & & \\
\text{(labels)} & n,m \in \omega \cup \{\infty\} & & \\
\text{(names)} & l,l' \in \mathcal{N} & & \\
\text{(variables)} & x,y,z \in \mathcal{X} & & \\
\text{(terms)} & a,b \in \mathcal{T} & ::= & x^n \mid \varepsilon^n \mid a^n \mid (\lambda x.a)^n \mid (a\,b)^n \mid \\
 & & & [l_i = \varsigma x.a_i{}^{i\in I}]^n \mid (a.l)^n \mid (a \Leftarrow l = \varsigma x.b)^n \\
\text{(hnf)} & h \in \mathcal{H} & ::= & x^{n+1} \mid (h\,a)^{n+1} \mid (h.l)^{n+1} \mid (h \Leftarrow l = \varsigma x.b)^{n+1} \\
\text{(values)} & v \in \mathcal{V} & ::= & h \mid (\lambda x.a)^{n+1} \mid ([l_i = \varsigma x.a_i{}^{i\in I}])^{n+1} \mid \varepsilon^{n+1}
\end{array}
$$

Fig. 1. Syntax

omitted. In consequence there is an obvious embedding of usual, unlabeled terms into labeled terms by decorating each subterm with ∞. Furthermore ∞ is considered the successor of itself, so by abuse of notation a superscript $n+1$ may denote ∞, in which case n also equals ∞.

Like in the lazy λ-calculus, every function or object is a value if its label is > 0; furthermore open terms in head normal form (i.e. starting with a free variable) are also values. Finally, notice that ε is a value, which is a bit uncommon, but is an essential point of the approach.

We adopt common conventions for simplifying notation: $\lambda xy.a$ for $\lambda x.\lambda y.a$, $(a\,b\,c)$ for $((a\,b)c)$. In an object $[l_i = \varsigma x.a_i{}^{i\in I}]$ it is implicitly understood that the order of methods is irrelevant, and that for $i,j \in I, l_i \neq l_j$ whenever $i \neq j$. A similar convention will be used for types in Section 4. A method $l = a$ in an object is an abbreviation for $l = \varsigma x.a$, where x does not occur free in a; in that case it is called a *field*. Some common terms are $\mathbf{I} = \lambda x.x, \mathbf{K} = \lambda xy.y, \Omega = (\lambda x.xx)(\lambda x.xx)$. A *context* $C[-]$ is a term possibly containing occurrences of a "hole"; $C[a]$ is the term obtained by filling the holes with a, with possible variable capture. A *substitution* σ is a finite map from variables to terms; $a\sigma$ is the term obtained by substituting free occurrences of x in a by $\sigma(x)$, while avoiding variable capture; a single substitution is written $a[x := b]$. The sets $\mathcal{T}^c$ and $\mathcal{V}^c$ are respectively the closed terms and the closed values. A *closing substitution* for a is a σ such that $a\sigma \in \mathcal{T}^c$. The set $\mathcal{T}^n$ is the set of terms with outermost label less or equal to n.

The one-step reduction relation $\to$ is the least relation satisfying the rules in Figure 2. Labels and errors are the two unusual factors in these rules. Labels are decremented at each step where a term is "deconstructed"; in case $n+1 = n = \infty$, i.e. when the counter is infinite, the rules just become the usual rules for reduction of functions and objects. Errors are a way to avoid

$$
\begin{array}{ll}
(\lambda\beta) & (\lambda x.a)^{n+1}b \to (a[x := b])^n \\
(\lambda\sigma) & (\lambda x.a)^{n+1}.l \to \varepsilon^n \\
(\lambda\upsilon) & (\lambda x.a)^{n+1} \Leftarrow l = \varsigma x.b \to \varepsilon^n \\
(o\sigma) & [l_i = \varsigma x.{a_i}^{i\in I}]^{n+1}.l_j \to \\
 & \qquad \begin{cases} (a_j[x := [l_i = \varsigma x.{a_i}^{i\in I}]^{n+1}])^n & \text{if } j \in I \\ \varepsilon^n & \text{otherwise} \end{cases} \\
(o\upsilon) & [l_i = \varsigma x.{a_i}^{i\in I}]^{n+1} \Leftarrow l_j = \varsigma x.b \to [l_i = \varsigma x.{a_i}^{i\in I\setminus\{j\}}, l_j = \varsigma x.b]^n \\
(o\beta) & [l_i = \varsigma x.{a_i}^{i\in I}]^{n+1}\, b \to \varepsilon^n \\
(\varepsilon\beta) & (\varepsilon^{n+1}\, a) \to \varepsilon^n \\
(\varepsilon\sigma) & \varepsilon^{n+1}.l \to \varepsilon^n \\
(\varepsilon\upsilon) & \varepsilon^{n+1} \Leftarrow l = \varsigma x.b \to \varepsilon^n \\
(\lambda\varepsilon) & (\lambda x.\varepsilon^m)^{n+1} \to \varepsilon^{1+\min(m,n)} \\
(\nu^0) & a^0 \to \Omega \\
(\nu) & (a^m)^n \to a^{\min(m,n)} \\
(\text{cong}) & a \to b \implies \forall C[-], C[a] \to C[b]
\end{array}
$$

Fig. 2. Reduction rules

so-called "stuck terms" in the literature: instead of having terms which do not reduce but are not values, we explicitly reduce them to the error constant. Once generated, errors are always propagated further in the computation, i.e. there is no exception handling construct; however, since this is a call-by-name calculus, a context may discard an error in the same way that it would discard a divergent subterm: for example $\mathbf{K}\varepsilon$ reduces to $\mathbf{I}$.

The $(\lambda\varepsilon)$ rule is an ad hoc rule which allows us to greatly simplify the abstract framework of [10]: instead of observing "ability to interact" we will just observe reduction to ε. Intuitively the rule is motivated by the fact that a function containing ε can do nothing "useful" and therefore is equivalent to ε. By contrast, there is no such rule for objects, because a method containing ε can always be overridden.

In this untyped calculus the $\Leftarrow$ operator can not only override existing methods, but also add new methods, which is more liberal than in [2]. This is a deliberate choice, so that the same calculus can be used to interpret various type systems. In the next section we start with the type system of [2], in which only override can be well-typed; later we extend it with the system of [17] which also supports method extension.

The k-transitive closure of $\to$ is written $\xrightarrow{k}$, its reflexive, transitive closure is written $\xrightarrow{*}$, and the symmetric closure of $\xrightarrow{*}$ is written $\overset{*}{=}$.

Theorem 2.1 (Confluence) *The language is confluent: whenever $a \to b$ and $a \to c$ there is a d such that $b \xrightarrow{*} d$ and $c \xrightarrow{*} d$.*

Proof. Standard Tait technique using parallel reductions; see for example [22,9]. □

Definition 2.2 [Convergence]

A term a *converges* ($a \Downarrow$) iff $\exists v \in \mathcal{V}, a \xrightarrow{*} v$. Otherwise a *diverges* ($a \Uparrow$).

3 Operational Subsumption

The idea of operational subsumption is a simulation relation based on observation of errors. In the abstract framework of [10] we had to build a complex machinery in order to define the notion of "erroneous terms". Here this can be much simpler: like in [9], we have a rule ($\lambda\varepsilon$) which removes a λ-abstraction if its body is an error; this rule is admissible because it does not break confluence (Theorem 2.1 above). As a result it suffices to observe reductions to ε as a basis for subsumption.

Definition 3.1 [Error terms]

$$a\dagger \iff \exists n, a \xrightarrow{*} \varepsilon^{n+1}$$

$\mathcal{E}$ will denote the set $\{a \mid a\dagger\}$ of error terms.

Definition 3.2 [Contextual subsumption]

A term a *contextually subsumes* another term b, written $a \sqsubseteq^{ctxt} b$, iff it generates fewer errors in all program contexts:

$$a \sqsubseteq^{ctxt} b \iff \forall C[-], C[a]\dagger \implies C[b]\dagger$$

Subsumption is a lattice with bottom Ω and top ε. The symmetric closure of $\sqsubseteq$ is written $\overset{\varepsilon}{=}$.

Lemma 3.3 *Subsumption contains reduction*

$$a \xrightarrow{*} b \implies a \overset{\varepsilon}{=} b$$

Proof. Direct from definition, knowing that the language is confluent. □

For convenience of proofs it is useful to establish a "context lemma" which allows us to only inspect a restricted set of contexts.

Definition 3.4 An *applicative context* $R[-]$ is a closed context generated by the following syntax:

$$R[-] \quad ::= \quad [-]^n \mid (R[-]\ a)^n \mid (R[-].l)^n \mid (R[-] \Leftarrow l = \varsigma x.a)^n$$

Definition 3.5 *Applicative subsumption* is the relation defined as

$$a \sqsubseteq^{appl} b \iff \forall \sigma, \forall R[-], R[a\sigma]\dagger \implies R[b\sigma]\dagger$$

Lemma 3.6 (i) $\lambda x.a \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b \implies b\dagger \vee (b \xrightarrow{*} \lambda x.b' \wedge a \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b')$

(ii) $a \equiv [l_i = \varsigma x.{a_i}^{i\in I}] \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b \implies b\dagger \vee (b \xrightarrow{*} [l_j = \varsigma x.{b_j}^{j\in J}], J \subseteq I \wedge \forall j \in J, a_j[x := a] \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b_j[x := b])$

Proof.

(i) Since $(\lambda x.a).l\dagger$ and $((\lambda x.a) \Leftarrow l = \varsigma x.c)\dagger$, b cannot reduce to an object. So either $b\dagger$, or $b \xrightarrow{*} \lambda x.b'$. Then $R[a[x := c]] \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} R[b'[x := c]]$ for every $R[-], c$, which implies $a \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b'$.

(ii) Similar reasoning.

□

Theorem 3.7 ("Ciu", context lemma)

$$a \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{ctxt} b \quad \iff \quad a \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b$$

Proof. The $\Rightarrow$ direction is trivial, since applicative contexts are contexts. The difficult part is the $\Leftarrow$ direction. We proceed by induction on the length of the proof of $C[a]\dagger$, i.e. we will show

$$\forall i, \forall C[-], ((C[a] \xrightarrow{i} \varepsilon^{n+1}) \wedge (a \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b)) \implies C[b]\dagger$$

The case $i = 0$ is trivial because then $C[-]$ is the empty context and both a and b are errors. If $i > 0$ and the first reduction step occurs either in $C[-]$ or in a, i.e. if $C[a] \to C'[a']$ with either $C[-] \to C'[-]$ or $a \to a'$, then we can directly use the induction hypothesis. Finally we can also ignore the cases where $b\dagger$, which again are trivial. So we are left with the following cases:

- cases $(\nu), (\nu^0)$: easy, a similar step can be performed with $C[b]$ and then we can appeal to the induction hypothesis.
- cases $(\varepsilon\beta), (\varepsilon\sigma), (\varepsilon\upsilon)$: easy again because both a and b must be error terms.
- cases $(\lambda\sigma), (\lambda\upsilon), (o\beta)$: these are the cases which generate an error. By the preceding Lemma a similar step can be performed with $C[b]$ and then the result follows from induction hypothesis.
- case $(\lambda\beta)$: here a must be of shape $\lambda x.a'$ and $C[-]$ must contain a subterm of shape $((-)B[-])$. Let $D[-], E[-]$ be the contexts repectively obtained by replacing this subterm by $(aB[-])$ or $(bB[-])$. Clearly $C[a] \equiv D[a]$ and $D[a] \to D'[a]$ where the same subterm is replaced by $a'[x := B[a]]$. We know $D'[a]\dagger$, and therefore by induction hypothesis $D'[b]\dagger$. Now by the preceding Lemma, $b \xrightarrow{*} \lambda x.b'$ with $a' \mathrel{\underline{\sqsubseteq}}_{\varepsilon}^{appl} b'$. Again by induction hypothesis, $E'[b]\dagger$, where $E'[b]$ is obtained from $D'[b]$ by replacing the same subterm by $b'[x := B[b]]$. Finally, since $E[b] \to E'[b]$ and $E[b] \equiv C[b]$, we have proved $C[b]\dagger$.
- cases $(o\sigma), (o\upsilon)$: like the preceding case, using the second clause of Lemma 3.6.

□

Because of this Theorem we will henceforth omit the ^{ctxt} or ^{appl} superscripts.

Example 3.8 Thanks to the preceding Theorem we can easily prove the following laws through induction on applicative contexts:

(i) $\lambda x_1 \ldots x_i.\varepsilon a_1 \ldots a_j \stackrel{\varepsilon}{=} \varepsilon$

(ii) $a \stackrel{\varepsilon}{\sqsubseteq} \lambda x.a\ x \quad$ if x is not free in a

(iii) $[l_i = \varsigma x.a_i{}^{i \in I}] \stackrel{\varepsilon}{\sqsubseteq} [l_j = \varsigma x.a_j{}^{j \in J}]$ if $J \subseteq I$

(iv) $\neg([l = a, l' = \varsigma x.\, x.l] \stackrel{\varepsilon}{\sqsubseteq} [l' = a])$

(v) $[l = \varsigma x.[l = \varsigma y.x]] \stackrel{\varepsilon}{=} [l = \varsigma x.x]$

(vi) $[l_i = \varsigma x.a_i{}^{i \in I}, l = \varepsilon] \stackrel{\varepsilon}{=} [l_i = \varsigma x.a_i{}^{i \in I}]$

(vii) $[] \stackrel{\varepsilon}{\neq} \varepsilon$

The first law is quite obvious. The second law is the familiar η-rule of the λ-calculus; here it is not an equality because η-reduction is always sound, while η-expansion is obviously not sound when a is an object. The third law is the basis for object subtyping: an object with more methods subsumes an object with fewer methods. The fourth law shows that objects are compared not only on the basis of field access, but also on field update: accessing field l' would yield the same result on both sides, but updating field l would make the two object incomparable. The fifth law is an example that is not covered by the equational system of [2], because it cannot be shown by a finite proof; this is similar to the problem of equivalence or subtyping of recursive types [4]. The last two laws are consequences of our design choice to also support method addition in the untyped calculus; if, instead, we had only method override, then the situations would be reversed (law 6 would be an inequality and law 7 would be an equality).

4 Inference of Abadi-Cardelli types

This section introduces a hybrid type system, inspired from both [2] and [18], which includes object types, bounded universal and existential quantification, and recursive types. Like in [18], this is an impredicative, implicitly typed system, so there are neither type abstraction constructs for universal types nor **pack/open** constructs for existential types; such types are introduced and eliminated in the inference rules without help from the term syntax. Similary, the isomorphisms between recursive types and their unfoldings are handled automatically in the subtyping rules, so there is no need for explicit **fold/unfold** constructs in the term syntax. Apart from these surface differences, the typable objects are the same as in [2], so for example this system can type method override, but not method addition (a more powerful system will be discussed in the next section). On the other hand the only significant difference with respect to [18] is the addition of subtyping and bounded quantification. The intersection and union types of [18] are not handled here, not because of any technical difficulty, but just because they are of limited interest in the current context.

Type syntax

$$T, U ::= \text{TOP} \mid X \mid T \to U \mid [l_i : T_i^{i \in I}]$$
$$\mid \forall(X <: T)U \mid \exists(X <: T)U \mid \mu X.T$$

A *basis* Γ is a list of statements of the form $x : T$ (type assumption) or $X <: T$ (subtype assumption). A basis is well-formed iff the subjects of all assumptions are distinct, and the subject X of a subtype assumption $X <: T$ does not appear free in T nor in any previous assumption. *Judgements* are of the form $\Gamma \vdash T <: U$ (subtyping judgement) or $\Gamma \vdash a : T$ (typing judgement). The subtyping rules are given in Figure 3. These are the same as [2], except for the *fold/unfold* rules which express the isomorphism between recursive types. Observe that object types support width subtyping, but no depth subtyping (the types of the methods are invariant).

Type inference rules are defined in Figure 4. Most rules are standard. The rules for introduction and elimination of quantified types are taken from [18], with some adaptations for accommodating bounds on the quantified type variable.

The type interpretation is very similar to what we did in [10], except that here we have second-order types and object types instead of record types. Types are interpreted as ideals in $\langle \mathcal{T}^c, \sqsubseteq \rangle$, i.e. non-empty, downward-closed subsets of closed terms. Let **Tset** denote the set of such subsets. Letters t, u, v range over **Tset**. For any $t \in$ **Tset**, t^n denotes the set $\{a^n | a \in t\}$ (finite projection). **Tset** is a lattice ordered by subset inclusion, with top element $\top = \mathcal{T}^c$ and bottom element $\bot = \{a \in \mathcal{T}^c \mid a \Uparrow\}$. Notice that so far neither $\top$ nor $\bot$ has a corresponding expression in the type syntax. A *type environment* η is a mapping from type variables to **Tset**. Given a type environment, a type interpretation function $\mathbf{Ti}[-]$ maps types to members of **Tset**. Figure 5 gives the type interpretation. Like in [10], we interpret recursive types through indexed families of type interpretations, following an idea of [8]; non-contractive type expressions, like for example $\mu X.X$, are naturally mapped to the bottom type [2].

Lemma 4.1 (i) $\forall T, \eta, \mathbf{Ti}[T]_\eta \in$ **Tset**.

(ii) *A type is* trivial *if its interpretation contains* ε^1. *If* η *does not map any type variable to a trivial type, then* $\mathbf{Ti}[T]_\eta$ *is non-trivial for any* T.

Proof. Induction on T. □

Definition 4.2 A type environment η *satisfies* a basis Γ, written $\eta \models \Gamma$, iff $\mathbf{Ti}[X]_\eta \subseteq \mathbf{Ti}[T]_\eta$ whenever $(X <: T) \in \Gamma$. Similarly, a closing substitution on terms σ *satisfies* a basis Γ and a type environment η, written $\sigma \models (\Gamma, \eta)$, iff $x\sigma \in \mathbf{Ti}[T]_\eta$ whenever $(x : T) \in \Gamma$. The notation $\eta, \sigma \models \Gamma$ abbreviates $\eta \models \Gamma \wedge \sigma \models (\Gamma, \eta)$

[2] for an explanation of contractive, see the literature on recursive types, e.g. [8,4,18,2]

$$(top)\frac{}{\Gamma \vdash T <: \mathrm{TOP}}$$

$$(refl)\frac{}{\Gamma \vdash X <: X}$$

$$(env)\frac{}{\Gamma, X <: T \vdash X <: T}$$

$$(arrow)\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash T_1 \to U_1 <: T_2 \to U_2}$$

$$(obj)\frac{}{\Gamma \vdash [l_i : {T_i}^{i \in I}, l_j : {U_j}^{j \in J}] <: [l_i : {T_i}^{i \in I}]}$$

$$(forall)\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma, X <: T_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall (X <: T_1) U_1 <: \forall (X <: T_2) U_2}$$

$$(exists)\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma, X <: T_1 \vdash U_1 <: U_2}{\Gamma \vdash \exists (X <: T_1) U_1 <: \exists (X <: T_2) U_2}$$

$$(rec)\frac{\Gamma, X <: Y \vdash T <: U}{\Gamma \vdash \mu X.T <: \mu Y.U}$$

$$(unfold)\frac{}{\Gamma \vdash \mu X.T <: T[X := \mu X.T]}$$

$$(fold)\frac{}{\Gamma \vdash T[X := \mu X.T] <: \mu X.T}$$

Fig. 3. Subtyping rules

Theorem 4.3 (Type soundness)

(i) $\Gamma \vdash T <: U \implies \forall \eta \models \Gamma, \mathbf{Ti}[T]_\eta \subseteq \mathbf{Ti}[U]_\eta$.

(ii) $\Gamma \vdash a : T \implies \forall \eta, \sigma \models \Gamma, a\sigma \in \mathbf{Ti}[T]_\eta$.

(iii) $\Gamma \vdash a : T \implies \neg(a\dagger)$.

Proof. 1) induction on the judgement $\Gamma \vdash T <: U$; 2) induction on the judgement $\Gamma \vdash a : T$; 3) direct from 2) and from the preceding Lemma. □

$$(sub)\frac{\Gamma \vdash a : T \quad \Gamma \vdash T <: U}{\Gamma \vdash a : U}$$

$$(var)\frac{}{\Gamma, x : T \vdash x : T}$$

$$(lam)\frac{\Gamma, x : T \vdash a : U}{\Gamma \vdash \lambda x.a : T \to U}$$

$$(appl)\frac{\Gamma \vdash a : T \to U \qquad \Gamma \vdash b : T}{\Gamma \vdash (a\ b) : U}$$

$$(obj)\frac{\forall i : \Gamma, x : T \vdash a_i : T_i \quad (T \equiv [l_i : T_i{}^{i \in I}])}{\Gamma \vdash [l_i = \varsigma x.a_i{}^{i \in I}] : T}$$

$$(sel)\frac{\Gamma \vdash a : [l_i : T_i{}^{i \in I}]}{\forall i \in I : \Gamma \vdash a.l_i : T_i}$$

$$(upd)\frac{\Gamma \vdash a : T \quad \Gamma, x : T \vdash b : T_k \quad (T \equiv [l_i : T_i{}^{i \in I}]) \quad k \in I}{\Gamma \vdash a \Leftarrow l_k = \varsigma x.b : T}$$

$$(\forall intro)\frac{\Gamma, X <: T \vdash a : U}{\Gamma \vdash a : \forall (X <: T) U}$$

$$(\forall elim)\frac{\Gamma \vdash a : \forall (X <: U) T \qquad \Gamma \vdash V <: U}{\Gamma \vdash a : T[X := V]}$$

$$(\exists intro)\frac{\Gamma \vdash a : T[X := U] \qquad \Gamma \vdash U <: V}{\Gamma \vdash a : \exists (X <: V) T}$$

$$(\exists elim)\frac{\Gamma \vdash a : \exists (X <: T) U \qquad \Gamma, X <: T, x : U \vdash b : V}{\Gamma \vdash b[x := a] : V}$$

Fig. 4. Inference rules

5 Variations

This section explores some variants of the type system to augment its expressive power. Thanks to the underlying untyped languages and to Theorem 3.7 the soundness of the new rules can be checked quite easily.

$$
\begin{aligned}
\mathbf{Ti}[T]^0_\eta &= \bot \\
\mathbf{Ti}[\mathrm{TOP}]^{n+1}_\eta &= \{a \in \mathcal{T}^{n+1} | \neg(a\dagger)\} \\
\mathbf{Ti}[X]^{n+1}_\eta &= \eta(X)^{n+1} \\
\mathbf{Ti}[T \to U]^{n+1}_\eta &= \{a \in \mathcal{T}^{n+1} | b \in \mathbf{Ti}[T]^n_\eta \implies (a\,b) \in \mathbf{Ti}[U]^n_\eta\} \\
\mathbf{Ti}[\,[l_i : T_i{}^{i\in I}]\,]^{n+1}_\eta &= \{a \in \mathcal{T}^{n+1} | a \sqsubseteq_\varepsilon [\,] \wedge \forall i \in I, a.l_i \in \mathbf{Ti}[T_i]^n_\eta\} \\
\mathbf{Ti}[\forall(X <: T)U]^{n+1}_\eta &= \bigcap_{t \subseteq \mathbf{Ti}[T]^{n+1}_\eta} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]} \\
\mathbf{Ti}[\exists(X <: T)U]^{n+1}_\eta &= \bigcup_{t \subseteq \mathbf{Ti}[T]^{n+1}_\eta} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]} \\
\mathbf{Ti}[\mu X.T]^{n+1}_\eta &= \mathbf{Ti}[T]^{n+1}_{\eta[X \mapsto \mathbf{Ti}[\mu X.T]^n_\eta]} \\
\\
\mathbf{Ti}[T]_\eta &= \{a | \forall n \in \omega, a^n \in \mathbf{Ti}[T]^n_\eta\}
\end{aligned}
$$

Fig. 5. Type interpretation

5.1 Super-top type

The system presented above is a "classical" subtyping system in the sense that there is a maximal type TOP containing all non-erroneous terms. If we had union types, TOP could be expressed as the union of all function and object types:

$$\mathrm{TOP} = \mu X.(X \to X) \cup [\,]$$

In [10] we have argued in favour of an even bigger type containing all terms, including erroneous ones. This can be easily incorporated into the system, by adding a new symbol $\top$ in the type syntax with interpretation $\mathbf{Ti}[\top]^{n+1}_\eta = (\mathcal{T}^c)^{n+1}$, and by adding the typing rule

$$(\top)\frac{}{\Gamma \vdash a : \top}$$

All previous results are preserved, except that for type soundness we have to exclude the set **Triv** of the *trivial* types generated by the following syntax:

$$Z \in \mathbf{Triv} ::= \top \mid T \to Z \mid \forall(X <: T)Z \mid \exists(X <: T)Z \mid \mu X.Z$$

Then the last point of Theorem 4.3 has to be reformulated as:

$$\Gamma \vdash a : T \wedge T \notin \mathbf{Triv} \implies \neg(a\dagger)$$

The interest of the $\top$ type is that the following subtyping rule is sound:

$$(obj\top)\frac{}{\Gamma \vdash [l_i : T_i^{i\in I}] <: [l_i : T_i^{i\in I}, l : \top]}$$

Hence a method with type $\top$ is equivalent to an absent method. We briefly repeat the example of [10] to show that this subtyping rule allows us to type more programs. Consider a translating function

$$\begin{aligned} T \stackrel{\text{def}}{=} \lambda x.[\,&\texttt{imprime} = x.\texttt{print},\\ &\texttt{affiche} = x.\texttt{display},\\ &\texttt{ferme} = x.\texttt{close}\ ] \end{aligned}$$

which takes an "english" object with three fields as argument, and returns a corresponding "french" object. Now consider object $O \stackrel{\text{def}}{=} [\texttt{display} = \text{"}\texttt{hello}\text{"}]$ and program $(T\,O).\texttt{affiche}$. This reduces to "`hello`" without error; however it cannot be typed in the original system because the type of T is

$$\begin{aligned} \forall X, Y, Z,\ &[\texttt{print} : X, \texttt{display} : Y, \texttt{close} : Z] \to\\ &[\texttt{imprime} : X, \texttt{affiche} : Y, \texttt{ferme} : Z] \end{aligned}$$

and $[\texttt{display} : \textbf{String}]$ (the type of O) cannot be unified with the left-hand side of the arrow. By contrast, the $\top$ extension allows us to infer

$$\emptyset \vdash O : [\texttt{print} : \top, \texttt{display} : \textbf{String}, \texttt{close} : \top]$$

and therefore the program $(T\,O).\texttt{affiche}$ has type **String**.

5.2 Diamond types

Liquori [17] proposed a type system for the object calculus which supports method extension. This is done through so-called *diamond types* of shape

$$[l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} \quad (\{l_i | i \in I\} \cap \{l_j | j \in J\} = \emptyset)$$

where the left part of the diamond expresses the available methods as usual, while the right part specifies the safe possible method extensions. The main subtyping rules are displayed in Figure 6.

Liquori establishes soundness of his system through a subject reduction theorem, showing that types are preserved during computation. In our framework soundness can be shown by proving that the typing rules agree with the type interpretation. Since we have method extension in the underlying untyped calculus, the addition and verification of diamond types is direct. We

$$(Shift\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I+K} <: [l_i : T_i \diamond l_j : T_j]_{j\in J+K}^{i\in I}}$$

$$(Extend\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} <: [l_i : T_i \diamond l_j : T_j]_{j\in J+K}^{i\in I}}$$

$$(Sat\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} <: [l_i : T_i]^{i\in I}}$$

Fig. 6. Diamond subtyping

interpret diamond types as

$$\begin{aligned}
&\mathbf{Ti}[\![[l_i : T_i^{i\in I} \diamond l_j : T_j^{j\in J}]]\!]_\eta^{n+1} = \\
&\quad \{a \in \mathbf{Ti}[\![[l_i : T_i^{i\in I}]]\!]_\eta^{n+1} \mid \forall l_j, \forall b, \\
&\qquad ((j \in J \wedge b[x := a] \in \mathbf{Ti}[T_j]_\eta^{n+1}) \vee j \notin J) \\
&\qquad\quad \Longrightarrow a{\Leftarrow}l_j = \varsigma x.b \in \mathbf{Ti}[\![[l_i : T_i^{i\in I\cup\{j\}}]]\!]_\eta^n\}
\end{aligned}$$

So this says that a method extension is safe if, whenever the field is mentioned in the right-hand side of the diamond, the body of the added method has a corresponding type. On the other hand if the field is not mentioned then there is no constraint on that field and the extension is safe in any case. Once this is understood the soundness of the subtyping rules for diamond types is easy to establish.

Lemma 5.1 *The subtyping rules of Figure 6 are sound.*

Proof. Omitted □

In addition we can easily verify other properties of diamond types, not mentioned in [17]. For example diamond types are contravariant on the right, i.e. the rule

$$(Contr\diamond)\frac{\forall j \in J, U_j <: T_j}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} <: [l_i : T_i \diamond l_j : U_j]_{j\in J}^{i\in I}}$$

is sound. Furthermore our "supertop" type $\top$ is compatible with diamond types and again is equivalent to absent fields, so the following rule is also sound:

$$(\top\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} = [l_i : T_i, l : \top \diamond l_j : T_j, l' : \top]_{j\in J}^{i\in I}}$$

Because of lack of space we do not repeat here the type inference rules of [17]; however it should be clear that soundness of these rules can be established by similar means, i.e. without need for a subject-reduction theorem.

6 Typed equivalences

In calculi with subtyping, the equivalence relationship between terms is dependent on the type context: for example the objects $[l_1 = 2, l_2 = 4]$ and $[l_1 = \varsigma x.(x.l_3), l_2 = \text{"}\texttt{foo}\text{"}, l_3 = 2]$ are equal at type $[l_1 : \mathbf{Int}]$, because the only possible observations at this type are on field l_1, where the two objects have the same value. This is precisely why types and subtypes are usually interpreted in partial equivalence relationships (PERs), which express exactly this relation; however PERs require denotational domains to be built from. In this section we show how this can be done in our operational framework through a type indexing of the subsumption relation.

Definition 6.1 [Relevant contexts] A context $C[-]$ is *relevant* iff $(a \Uparrow \implies C[a] \Uparrow)$ and $C[\varepsilon] \Downarrow$. The set of relevant contexts is written $\mathcal{C}$.

The idea here is that a context is irrelevant when no observation can be made about the term filling the hole. The first condition ensures that divergence at the hole is propagated to the outer level. The second condition rules out the contexts which are always divergent, because these also hide any observation from the hole.

Lemma 6.2

$$C[-] \in \mathcal{C} \implies C[\varepsilon]\dagger$$

Proof. By contradiction, showing that if $\neg(C[\varepsilon]\dagger)$ then $\neq (C[-] \in \mathcal{C})$. This is done by induction on the length of the reduction $C[\varepsilon] \xrightarrow{*} v$, comparing at each step with $C[\Omega]$. □

Definition 6.3 [Type-dependent Contexts]

$$\mathcal{C}_t \equiv \{C[-] \in \mathcal{C} | \forall a \in t, \neg(C[a]\dagger)\}$$

Lemma 6.4 (i) $\mathcal{C}_\bot = \mathcal{C}$

(ii) $\mathcal{C}_\top = \emptyset$

(iii) $\mathcal{C}_{\mathcal{T}\setminus\mathcal{E}} = \{C[-] | \forall a, \exists n, C[a] \xrightarrow{*} a^{n+1}\}$

(iv) $t \subseteq u \implies \mathcal{C}_u \subseteq \mathcal{C}_t$

Proof.

(i) for every relevant context $C[-]$ and divergent term $a \in \bot$, $C[a] \Uparrow$ and therefore $\neg(C[a]\dagger)$.

(ii) $\varepsilon \in \top$ and every relevant context filled with ε is erroneous (Lemma 6.2).

(iii) $\mathcal{T} \setminus \mathcal{E}$ contains both objects and functions. Therefore $C[-]$ cannot contain a subterm of shape $([-]B[-])$, because the hole could be filled by an object and the β reduction would yield an error. Conversely, the hole cannot either participate in a σ or υ reduction. Therefore the only reductions involving the hole must be ν reductions. Finally $C[-]$ cannot take a to a^0 because then it would be irrelevant.

(iv) If $C[-] \in \mathcal{C}_u$ then $\forall a \in u, \neg(C[a]\dagger)$; but then $\forall a \in t, \neg(C[a]\dagger)$ because $t \subseteq u$; therefore $C[-] \in \mathcal{C}_t$.

□

Definition 6.5 [Type-dependent subsumption]

$$a \mathrel{\underline{\sqsubseteq}}_t b \iff (a, b \in t \wedge (\forall C[-] \in \mathcal{C}_t, C[a]\dagger \implies C[b]\dagger))$$

Lemma 6.6 (i) $(\mathrel{\underline{\sqsubseteq}}_\bot) = (\mathrel{\underline{\sqsubseteq}})$

(ii) $(\mathrel{\underline{\sqsubseteq}}_\top) = \mathcal{T} \times \mathcal{T}$

(iii) $(\mathrel{\underline{\sqsubseteq}}_{\mathrm{TOP}}) = (\mathcal{T} \setminus \mathcal{E}) \times (\mathcal{T} \setminus \mathcal{E})$

(iv) $t \subseteq u \implies ((\mathrel{\underline{\sqsubseteq}}_t) \subseteq (\mathrel{\underline{\sqsubseteq}}_u))$

Proof. Direct consequences of Lemma 6.4. □

Now we overload the notation to define a family of subsumption relations indexed by syntactic types (as opposed to the semantic types used above).

Definition 6.7 [Syntactic type-dep. subsumption]

$$a \mathrel{\underline{\sqsubseteq}}^n_{T,\eta} b \iff a \mathrel{\underline{\sqsubseteq}}_{\mathbf{Ti}[T]^n_\eta} b$$

Lemma 6.8 (i) $\forall a, b \in \mathcal{T}^{n+1}, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\top,\eta} b$

(ii) $\forall a, b \in \mathcal{T}^{n+1} \setminus \mathcal{E}, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\mathrm{TOP},\eta} b$

(iii) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{T\to U,\eta} b \iff \forall c \in \mathbf{Ti}[T]^n_\eta, (ac) \mathrel{\underline{\sqsubseteq}}^n_{U,\eta} (bc)$

(iv) $Let\, T \equiv [l_i : T_i^{\,i\in I}]\,.\, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{T,\eta} b \iff \forall i \in I,$

$a.l_i \mathrel{\underline{\sqsubseteq}}^n_{T_i,\eta} b.l_i \wedge$

$(c[x := b] \in \mathbf{Ti}[T_i]^{n+1}_\eta \implies$

$(a \Leftarrow l_i = \varsigma x.c) \mathrel{\underline{\sqsubseteq}}^n_{T,\eta} (b \Leftarrow l_i = \varsigma x.c))$

(v) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\forall(X<:T)U,\eta} b \iff \forall t \subseteq \mathbf{Ti}[T]^{n+1}_\eta, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{U,\eta[X\mapsto t]} b$

(vi) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\exists(X<:T)U,\eta} b \iff \exists t \subseteq \mathbf{Ti}[T]^{n+1}_\eta, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{U,\eta[X\mapsto t]} b$

(vii) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\mu X.T,\eta} b \iff a \mathrel{\underline{\sqsubseteq}}^{n+1}_{T[X:=\mu X.T],\eta} b$

Proof. Double induction on the shape of types and on the index n. □

Definition 6.9 [Typed equalities] The interpretation of typed equalities is:

$$\mathbf{Ti}[\Gamma \vdash a \leftrightarrow b : T] \quad = \quad \forall \eta, \sigma \models \Gamma, \forall n, (a\sigma \mathrel{\overset{\varepsilon}{=}}^n_{T,\eta} b\sigma)$$

Notice that this interpretation accounts for open objects and open types.

Lack of space prevents us from checking the complete set of equational rules of [2]. However it is worth noticing that for most of them we can take advantage of Lemma 6.6: if we are able to prove $a \overset{\varepsilon}{=} b$ then $a \leftrightarrow b : T$ at any T. This in particular covers all "Eval" rules of [2], because $a \overset{*}{\to} b \implies a \overset{\varepsilon}{=} b$ (Property 3.3). So we will concentrate on a few interesting rules that are more specifically related to subtyping. These are displayed in Figure 7. The first

$$\text{(Eq Sub Obj)}\ \frac{T \equiv [l_i : T_i^{\,i\in I}] \quad T' \equiv [l_i : T_i^{\,i\in I\cup J}] \qquad \Gamma, x : T \vdash a_i : T_i \quad \forall i \in I \qquad \Gamma, x : T' \vdash a_j : T_j \quad \forall j \in J}{\Gamma \vdash [l_i = \varsigma x.a_i^{\,i\in I}] \leftrightarrow [l_i = \varsigma x.a_i^{\,i\in I\cup J}] : T}$$

$$\text{(Eq } \forall \text{ Intro)}\frac{\Gamma, X <: T \vdash a \leftrightarrow b : U}{\Gamma \vdash a \leftrightarrow b : \forall(X <: T)U}$$

$$\text{(Eq } \forall \text{ Elim)}\frac{\Gamma \vdash a \leftrightarrow b : \forall(X <: T)U \quad \Gamma \vdash T' <: T}{\Gamma \vdash a \leftrightarrow b : U[X := T']}$$

$$\text{(Eq } \exists \text{ Intro)}\frac{\Gamma \vdash a \leftrightarrow b : U[X := T'] \quad \Gamma \vdash T' <: T}{\Gamma \vdash a \leftrightarrow b : \exists(X <: T)U}$$

$$\text{(Eq } \exists \text{ Elim)}\frac{\Gamma \vdash a \leftrightarrow b : \exists(X <: T)U \quad \Gamma, X <: T, x : U \vdash c \leftrightarrow d : V}{\Gamma \vdash c[x := a] \leftrightarrow d[x := b] : V}$$

Fig. 7. Some equational rules

one is taken from [2]. The other rules have to do with second-order types and subtyping; similar rules can be found in the literature for explicitly typed systems (system $F_<$), but to the best of our knowledge the present formulation for implicitly typed systems is new.

Theorem 6.10 *The rules of Figure 7 are sound.*

Proof. (Eq Sub Obj): one direction comes directly from the untyped subsumption (Example 3.8, item 3). For the other direction, it suffices to observe that, for any η, contexts in $\mathcal{C}_{\mathbf{Ti}[T]_\eta}$ can only use names in $\{l_i | i \in I\}$. ($\forall$ and $\exists$ rules): first observe that $\forall \eta, \mathbf{Ti}[U[X := T]]_\eta = \mathbf{Ti}[U]_{\eta[X \mapsto \mathbf{Ti}[T]_\eta]}$; second, for any logical predicate $P(\eta)$ we have the following equivalence:

$$\forall \eta \models (\Gamma, X <: T).P(\eta) \equiv \forall \eta' \models \Gamma. \forall t \subseteq \mathbf{Ti}[T]_{\eta'}.P(\eta'[X \mapsto t])$$

Taking advantage of these facts, for example if we take $P(\eta) \equiv \forall n, a \stackrel{\varepsilon}{=}^n_{U,\eta} b$ we immediately get a soundess proof for for (Eq $\forall$ Intro). Other rules are handled similarly. □

7 Structural subtyping

In the previous section, subtyping was interpreted as set inclusion. This is quite close in spirit to the two other interpretations found in the literature, namely coercion functions [6] or inclusions between partial equivalence relations (PERs) [7]. However it also suffers from the same problem, first explained in [7]: the bounded polymorphic type

$$\forall(X <: T)X \to X$$

can only contain the identity function. Intuitively this comes from the fact that for every member a of T there is a semantic subtype of T containing only that single member, and then the function can do nothing but map that member to itself. This is annoying because a function like

$$\lambda x.x \Leftarrow l = \varsigma y.\mathbf{not}(x.l)$$

cannot be assigned type

$$\forall X <: [l : \mathbf{Bool}].X \to X$$

Even if there is no syntactic type $[l : \mathbf{True}]$, there is an ideal $\{a \mid a.l \sqsubseteq^{\varepsilon} \mathbf{true}\}$ contained in **Bool** at which the specification $X \to X$ is unsound. In consequence there is no way to express in the type that this function leaves all fields other than l untouched.

Several approaches have been taken to fix the problem. As already noted in [7], the problem comes from the fact that the semantics contains "too many subtypes", typically many more than can be defined in the type syntax. One possibility then is to drop the denotational semantics altogether. Chapter 16 of [2] takes such an approach: it introduces stronger rules called "structural rules" which take advantage of the fact that all operations on objects preserve some implicit structure; these rules are unsound in the denotational semantics, but are proved to be operationally sound. Another possibility is to fix the semantic interpretation of subtyping. This program is carried out in [20], where subtyping is restricted to pointwise equality on fields between "record PERs"; however this interpretation prohibits depth subtyping on records, which is somewhat unsatisfactory because intuitively this form of subtyping is structural. Hofmann and Pierce [13] have a more general approach where the statement $T <: U$ is interpreted as a standard coercion $c : T \to U$ together with an overwrite function $put[T, U] : T \to U \to U$ which updates the "U part" of an element of T without changing the "remaining part". However this approach only works in cases of "positive subtyping", since there is no function dual to $put[T, U]$ to go down in the type hierarchy.

7.1 Structural subtyping as embedding-projections

Here we propose a new interpretation of subtyping by embedding-projection pairs, very similar in spirit to the maps used in domain theory for solving recursive equations through inverse limit constructions. The embedding $\uparrow_t^u : t \to u$ from a subtype to its supertype is just an inclusion map, so in the following there is no need to mention it explicitly. By contrast the projection $\downarrow_t^u : u \to t$ has to "preserve structure": the image must be equal to the argument at type u.

Definition 7.1 [structural inclusion]

$$t \subseteq\!\!\downarrow u \iff t \subseteq u \wedge \exists \downarrow_t^u : u \to t, \forall a \in u, \downarrow_t^u (a) = \min(\{b \in t | b \doteq_u^{\varepsilon} a\})$$

The essential condition here is that for every $a \in u$ there must be a $b \in t$

$$(struct-forall)\frac{\Gamma, X <\!\!\downarrow T \vdash U_1 <\!\!\downarrow U_2}{\Gamma \vdash \forall(X <\!\!\downarrow T)U_1 <\!\!\downarrow \forall(X <\!\!\downarrow T)U_2}$$

$$(struct-exists)\frac{\Gamma, X <\!\!\downarrow T \vdash U_1 <\!\!\downarrow U_2}{\Gamma \vdash \exists(X <\!\!\downarrow T)U_1 <\!\!\downarrow \exists(X <\!\!\downarrow T)U_2}$$

Fig. 8. Structural subtyping for quantified types

such that $b \stackrel{\varepsilon}{=}_u a$. The canonical choice of the minimal element is a secondary condition which proves to be convenient for dealing with quantified types.

Before proceeding with our calculus it is worth considering informally why subtyping is not always structural. Consider examples such as $[1 \ldots 10] <: [1 \ldots 20]$ or **True** $<:$ **Bool**. These make sense as subset inclusion, but the only way to map every element of the supertype to an element of the subtype is the constant function $\lambda x.\Omega$, which loses all information about its argument; therefore the "structure" of the supertype is lost. Similarly, the rules $T \cap U <: T$ or $\bot <: T$ in systems with intersection types or bottom types are non-structural. Finally, since universal and existential quantification are interpreted as intersection and union, the subtyping rules *forall* and *exists* in Figure 3 also break structure. Therefore it is not possible to just keep the system of Section 4 and reinterpret $<:$ as $\subseteq\!\!\downarrow$.

A system involving *both* subtyping relations is perfectly conceivable: it suffices to add a distinction in the type syntax between ordinary subtyping statements $T <: U$ and structural statements $T <\!\!\downarrow U$. Then in addition to the ordinary quantified types we also would have structurally quantified types $\forall(X <\!\!\downarrow T)U$ and $\exists(X <\!\!\downarrow T)U$, with the obvious interpretation

$$\mathbf{Ti}[\forall(X <\!\!\downarrow T)U]^{n+1}_{\eta} = \bigcap_{t \subseteq\!\!\downarrow \mathbf{Ti}[T]^{n+1}_{\eta}} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]}$$

$$\mathbf{Ti}[\exists(X <\!\!\downarrow T)U]^{n+1}_{\eta} = \bigcup_{t \subseteq\!\!\downarrow \mathbf{Ti}[T]^{n+1}_{\eta}} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]}$$

However even if the semantic apparatus is ready, it is not clear whether having two distinct notions of subtyping simultaneously is a desirable feature. The advantage of structural subtyping is that it validates some more powerful typing rules, like the (*val structural update*) rule shown below. On the other hand it forbids some "atomic subtyping" rules such as **True** $<:$ **Bool** or **Posint** $<:$ **Int**. As discussed here, having both is possible, but at the cost of a greater complexity in the type system. Choosing the most appropriate solution is a matter of language design. Here we will briefly discuss a system with structural subtyping only.

Definition 7.2 [Structural subtyping] The system of structural subtyping defines judgements of shape $\Gamma \vdash T <\!\!\downarrow U$; it is obtained from the system of

Section 4 by

(i) replacing subtyping assumptions $X <: T$ in Γ by structural subtyping assumptions $X <\!\!\downarrow T$;

(ii) replacing $<:$ by $<\!\!\downarrow$ in the rules of Figure 3;

(iii) replacing rules (*forall*) and (*exists*) in Figure 3 by the weaker version given in Figure 8.

For this system we obviously reinterpret the statement $\eta \models \Gamma$ as $\mathbf{Ti}[X]_\eta \subseteq\!\!\downarrow \mathbf{Ti}[T]_\eta$ whenever $(X <\!\!\downarrow T) \in \Gamma$.

Theorem 7.3 (Soundness of structural subtyping)

$$\Gamma \vdash T <\!\!\downarrow U \implies \forall \eta \models \Gamma, \mathbf{Ti}[T]_\eta \subseteq\!\!\downarrow \mathbf{Ti}[U]_\eta$$

Proof. Induction on the proof of $\Gamma \vdash T <\!\!\downarrow U$. For each subtyping rule used we exhibit the witness function $\downarrow^{\mathbf{Ti}[U]_\eta}_{\mathbf{Ti}[T]_\eta}$, which will be abbreviated as $\downarrow^U_T$.

- case (*top*): $\downarrow^{TOP}_T = \lambda x.\Omega$.
- case $(refl)$: $\downarrow^X_X = \lambda x.x$.
- case (env): from the assumption $\eta \models \Gamma, X <\!\!\downarrow T$ there exists a projection $\downarrow^{\mathbf{Ti}[T]_\eta}_{\mathbf{Ti}[X]_\eta}$.
- case $(arrow)$: $\downarrow^{T_2 \to U_2}_{T_1 \to U_1} = \lambda f.\, \downarrow^{U_2}_{U_1} \circ f \circ \downarrow^{T_1}_{T_2}$.
- case (obj): $\downarrow^{[l_i : T_i{}^{i \in I}]}_{[l_i : T_i{}^{i \in I \cup J}]} = \lambda x.x \Leftarrow l_{j_1} = \Omega \ldots \Leftarrow l_{j_k} = \Omega \quad (J = \{j_1 \ldots j_k\})$
- case $(forall)$: we know that $\forall \eta, \forall t \subseteq\!\!\downarrow \mathbf{Ti}[T]_\eta, \exists \downarrow^{\mathbf{Ti}[V]_{\eta[X \mapsto t]}}_{\mathbf{Ti}[U]_{\eta[X \mapsto t]}}$ Therefore we can build

  $$\downarrow^{\forall (X <\!\!\downarrow T) V}_{\forall (X <\!\!\downarrow T) U} = \bigcap_{t \subseteq\!\!\downarrow \mathbf{Ti}[T]_\eta} \downarrow^{\mathbf{Ti}[V]_{\eta[X \mapsto t]}}_{\mathbf{Ti}[U]_{\eta[X \mapsto t]}}$$

  By the fact that each projection function canonically chooses the minimal element in its codomain, their intersection is well-defined and satisfies the conditions of Definition 7.1.
- case $(exists)$: like the preceding case, taking $\bigcup$ instead of $\bigcap$.
- case (rec): using the premice $\Gamma, X <\!\!\downarrow Y \vdash T <\!\!\downarrow U$, we show by induction on n that the family of projections $\downarrow^{\mathbf{Ti}[\mu Y.U]^n_\eta}_{\mathbf{Ti}[\mu X.T]^n_\eta}$ is well-defined. Then the projection $\downarrow^{\mu Y.U}_{\mu X.T}$ is the union of such projections.
- cases $(fold), (unfold)$: $\downarrow^{\mu X.T}_{T[X:=\mu X.T]} = \downarrow^{T[X:=\mu X.T]}_{\mu X.T} = \lambda x.x$

□

Corollary 7.4 *The "val structural update" rule of [2] is sound:*

$$\frac{\Gamma \vdash a : T \quad \Gamma \vdash T <\!\!\downarrow [l_i : T_i{}^{i \in I}] \quad \Gamma, x : T \vdash b : T_j \quad j \in I}{\Gamma \vdash a \Leftarrow l_j = \varsigma x.b : T}$$

Proof. Suppose $\eta, \sigma \models \Gamma$ and $b\sigma[x := a] \in \mathbf{Ti}[T_j]_\eta$. Let $a' \equiv a \Leftarrow l_j = \varsigma x.b$ and $a'' \equiv \downarrow_T^{[l_i:T_i{}^{i\in I}]}(a')$: we must have $a'' \in \mathbf{Ti}[T]_\eta$ and $a'' \stackrel{\varepsilon}{=}_{[l_i:T_i{}^{i\in I}],\eta} a'$. Hence T is closed under updates of type T_j at l_j. □

Like [20], we now show an isomorphism which proves that type $\forall(X <\downarrow [l : T])X \to X$ contains more functions than just the identity.

Theorem 7.5 *The types $\forall(X <\downarrow [l : T])X \to X$ and $T \to T$ are isomorphic.*

Proof. Consider functions

$$F = \lambda f.\lambda x.(f([l = x]))\,.\,l$$

$$G = \lambda f.\lambda x.x \Leftarrow l = f(x.l)$$

It is easy to see that $F \circ G = G \circ F = \mathbf{I}$, and that $F : (\forall(X <\downarrow [l : T])X \to X) \to (T \to T)$. By contrast structural subtyping is required for the reverse typing: $G : (T \to T) \to (\forall(X <\downarrow [l : T])X \to X)$ is only derivable if we have the *(val structural update)* rule above. □

8 Conclusion

Thanks to several recent works, operational techniques are regaining considerable interest. Results such as fixpoint induction, which once were only provable through denotational means, are now shown with operational bisimilarities [21]. However in these works subtyping was seldom taken into account. The framework proposed in [10], introducing an explicit error constant ε which becomes the top element of the semantic lattice, can remedy to this deficiency. By applying it here to the object calculus of Abadi and Cardelli, we have demonstrated that some complex aspects of the abstract framework (namely the defintion of erroneous terms) can be greatly simplified when working with a concrete calculus. Furthermore we have introduced a number of technical innovations or improvements to previous work:

- rules for bounded second-order types in an implicitly typed system;
- an operational interpretation of typed equalities which deals with open terms and open types;
- a semantic interpretation of "structural subtyping";
- a "super-top" type which improves the typing power of the system without impairing type soundness.

Acknowledgements

A previous draft of this paper received very detailed comments from anonyous referees; these were extremely useful to improve the quality of the paper. Comments from various participants to the HOOTS II workshop are also gratefully acknowledged.

References

[1] Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Information and Computation*, 105:159-267, 1993.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Monographs in Computer Science, 1996.

[3] Martin Abadi, Benjamin Pierce and Gordon Plotkin. Faithful Ideal Models for Recursive Polymorphic Types. *Int. J. of Foundations for Computer Science*, 2(1):1-21, 1991.

[4] Roberto Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Trans. on Prog. Lang and Systems*, 15(4):575-631, 1993.

[5] Henk Barendregt. *The Lambda-Calculus, its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[6] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as Implicit Coercion. *Information and Computation* 93:172-221, 1991. Also in [12], pp 197-245.

[7] Kim Bruce and Giuseppe Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation* 87:196-240, 1990. Also in [12], pp 151-195.

[8] Felice Cardone and Mario Coppo. Two extensions of Curry's Type Inference System. In *Logic and Computer Science*, P. Odifreddi(ed), pp 19-75. Academic Press, 1990.

[9] Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Comp. Sc.* 192(2):201-231, Feb 1998.

[10] Laurent Dami. Labeled Reductions, Runtime Errors, and Operational Subsumption. Extended abstract in *Proc. ICALP'97*, LNCS, Springer-Verlag, 1997. Expanded version in *Objects at Large*, Technical Report, University of Geneva, 1997, available from `http://cuiwww.unige.ch/~dami/publis.html`.

[11] Andrew Gordon and Gareth Rees. *Bisimilarity for a First-Order Calculus of Objects with Subtyping*. Technical Report 386, Computer Laboratory, University of Cambridge, January 1996, available at `http://www.cl.cam.ac.uk/ adg`. Technical summary in *Proceedings 23rd ACM POPL*, Jan 1996, pp 386-395.

[12] Carl A. Gunter and John C. Mitchell, eds. *Theoretical aspects of object-oriented programming: types,semantics, and language design*. MIT Press, Foundations of computing series, 1994.

[13] Martin Hofmann and Benjamin C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11-33, 10 April 1996.

[14] D. J. Howe. Equality in lazy computation systems. In *Proc. 4th IEEE Symp. on Logic in Comp. Sc.*, pp 198-203, 1989.

[15] Trevor Jim and Albert R. Meyer. Full Abstraction and the Context Lemma. *SIAM J. on Computing* 25(3):663-696, June 1996.

[16] Marina Lenisa. Semantic Techniques for Deriving Coinductive Characterizations of Observational Equivalences for Lambda-Calculi. Proc. *TLCA'97*, pp 248-266. LNCS 1210, Springer-Verlag, 1997.

[17] Luigi Liquori. An Extended Theory of Primitive Objects: First Order System. *Proc. ECOOP'97*, pp 146-169, LNCS 1241, Springer-Verlag, 1997.

[18] David MacQueen, Gordon Plotkin and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. *Information and Control*, 71:95-130, 1986.

[19] Ian A. Mason, Scott F. Smith and Carolyn L. Talcott. From Operational Semantics to Domain Theory. *Information and Computation*, 128:26-47, 1996.

[20] Erik Poll. System F with Width-subtyping and Record Updating. *Proc. Internat. Symp. on Theoret. Aspects of Comp. Software*, Sendai, Japan. LNCS, Springer-Verlag, 1997.

[21] Scott Smith. The Coverage of Operational Semantics. In *Higher Order Operational Techniques in Semantics*, A. Gordon and A. Pitts, editors, Cambridge University Press, 1998.

[22] M. Takahashi. Parallel Reductions in λ-calculus. *Information and Computation*, 118(1):120-127, 1995.